Alberto Paoluzzi and Giorgio Scorzelli

BIM geometry with Julia Plasm

Functional language for CAD programming

June 14, 2024

Springer Nature

1	3
Pre/Post conds	4 → 5
Example	6

Contents

Chapter 5
Symbolic modeling with Julia Plasm

Symbolic modeling is a semantic approach to knowledge representation and processing. A symbolic approach to design with the aim of representing information and computation uses names to define the meaning of represented knowledge explicitly. The geometric knowledge is described here by Julia's names, which are chosen suitably for functionals, functions, formal and actual parameters, and finally for objects, fields, classes, attributes, methods, relations, etc. In this chapter, we give many examples of high-level Plasm programming, from topological, linear, and affine operators, to geometric mapping of complexes and grids to generate linearized approximation of curved manifold of intrinsic dimensions 1, 2, and 3. i.e., depending on such number of parameters; say, curves, surfaces, thin, and bulk solids.

5.1 Primitive generators

Here, we introduce both single objects and aggregates of cells, typically by grid and mesh generators, resulting in a single `Hpc` value after the evaluation.

Higher order and partial functions

As we have already seen in Section **??**, Julia `Plasm` is higher-level since allows for function that take functions as argument and/or may return a function value. All functions are objects of Julia `Function` type. As objects (holding a reference to the function code), can be assigned to a name (identifier).

Definition 5.1 (Function order) The *order* of an object of `Function` type is the number of applications to actual parameters needed to return the ultimate actual value, not a partial function value (needing further parameters).

Coding 5.1.1 *(*INTERVALS(size::Numrber)(n::Int)) In this example we show a second-order function (requiring two applications) that generates a 1D complex made by `n` line segments of total given `size` length.

```
segments = INTERVALS(10::Number)(4::Int)        #=
Hpc(MatrixNd(2), Geometry([[0.0], [2.5], [5.0], [7.5], [10.0]],
    hulls=[[1, 2], [2, 3], [3, 4], [4, 5]]))        =#
```

Note that `segments` value is 1D since its 11 vertices have one coordinate. □

Coding 5.1.2 *(*QUOTE(measures::ArrayNumber)) The formal parameter is an array of signed numbers.

```
two_aligned_segments = QUOTE([1,-2.5,1])          #=
Hpc(MatrixNd(2), Geometry([[0.0], [1.0], [3.5], [4.5]], hulls
    =[[1, 2], [3, 4]]))    =#
```

Positive numbers denote solid intervals of a given size; negative numbers denote hollow space, i.e., displacement of following segments. Successive negative numbers are allowed. □

Coding 5.1.3 *(*Q(measure::Number)) The formal parameter is a signed number.

```
segment = Q(10)
Hpc(MatrixNd(2), Geometry([[0.0], [10.0]], hulls=[[1, 2]]))
```

A single `segment` of given size. □

Single convex cell

Julia `Plasm` contains a great library of generator functions of very simple objects made by a single convex cell, and completely specified by its set of vertices only. Few examples follow; Other examples can be extracted or generated by the user looking at file `src/fenvs.jl`, including the Platonic solids. The multidimensional d-permutaheron is generated in Coding **??**.

Coding 5.1.4 *(*CUBOID(size)) Multidimensional cuboid with `sizes::Vector{Number}`.

```
CUBOID([1])        #=
Hpc(MatrixNd(2), Hpc(MatrixNd(2), Geometry([[0.0], [1.0]], hulls
    =[[1, 2]]))) =#

CUBOID([1,2])          #=
Hpc(MatrixNd([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0,
    2.0]]), Hpc(MatrixNd(3), Geometry([[0.0, 0.0], [1.0, 0.0],
    [1.0, 1.0], [0.0, 1.0]], hulls=[[1, 2, 3, 4]]))) =#
```

```
CUBOID([1,2,3])        #=
Hpc(MatrixNd([[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0], [0.0,
    0.0, 2.0, 0.0], [0.0, 0.0, 0.0, 3.0]]), Hpc(MatrixNd(4),
    Geometry([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0], [0.0, 1.0, 0.0],
     [1.0, 1.0, 0.0], [0.0, 0.0, 1.0], [1.0, 0.0, 1.0], [0.0,
    1.0, 1.0], [1.0, 1.0, 1.0]], hulls=[[1, 2, 3, 4, 5, 6, 7,
    8]]))) =#

CUBOID([1,2,3,4])        #=
Hpc(MatrixNd([[1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0,
    0.0], [0.0, 0.0, 2.0, 0.0, 0.0], [0.0, 0.0, 0.0, 3.0, 0.0],
    [0.0, 0.0, 0.0, 0.0, 4.0]]), Hpc(MatrixNd(5), Geometry
    ([[0.0, 0.0, 0.0, 0.0], [1.0, 0.0, 0.0, 0.0], [0.0, 1.0,
    0.0, 0.0], [1.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [1.0,
     0.0, 1.0, 0.0], [0.0, 1.0, 1.0, 0.0], [1.0, 1.0, 1.0, 0.0],
     [0.0, 0.0, 0.0, 1.0], [1.0, 0.0, 0.0, 1.0], [0.0, 1.0, 0.0,
     1.0], [1.0, 1.0, 0.0, 1.0], [0.0, 0.0, 1.0, 1.0], [1.0,
    0.0, 1.0, 1.0], [0.0, 1.0, 1.0, 1.0], [1.0, 1.0, 1.0, 1.0]],
     hulls=[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16]]))) =#
```

Cuboids of given sizes. Of course, the unit hypercube in $\mathbb{E}^6$ has `size = [1,1,1,1,1,1]`. □

The `Plasm` coding of the "icosphere", polyhedral approximation of the 2-sphere obtained by subdividing the `ICOSAHEDRON()` surface is given here, starting from the Platonic solid. The generation method is extremely simple. We obtain the vertices at step $i+1$ by adding to the vertices at step i those obtained by subdivision of all edges. Ma make use of theHpc structure and the Lar structure.

Coding 5.1.5 *(*ICOSPHERE(seed::Hpc)::Hpc) First we generate the cell complex of the input `obj` using the `LAR` combinator, then for each edge we compute the mean point, then we aggregate to the old vertices the new ones, scaled by the factor `r1/s1` built with the distance from `[0,0,0]` center of both models.

```
function ICOSPHERE(obj::Hpc)::Hpc
   W  = LAR(obj).V
   EV = LAR(obj).C[:EV]
   W  = [W[:,k] for k=1:size(W,2)]
   V  = [(W[v1]+W[v2])./2 for (v1,v2) in EV]
   r1 = sqrt(sum(W[1].^2))
   s1 = sqrt(sum(V[1].^2))
   CONVEXHULL([W; V*(r1/s1)]);
end
```

Finally, the `[W; V*(r1/s1)]::Vector{Vector{Float64}}` made by old vertices and by new scaled ones is given to the operator `CONVEXHULL` that transforms such a `Vector` of point (`Vector{Float64}`) in their geometric *convex* *hull*. Just remember that such polyhedra are convex sets, hence they have a *single* (convex) cell.

```
out0 = ICOSAHEDRON();   VIEW(out0)
out1 = ICOSPHERE(out0); VIEW(out1)
out2 = ICOSPHERE(out1); VIEW(out2)
out3 = ICOSPHERE(out2); VIEW(out3)
...
```

Successive approximations of icosphere with 12, 42, 162, 600, etc., vertices. Let's remark the extreme simplicity of such polyhedral generations. □

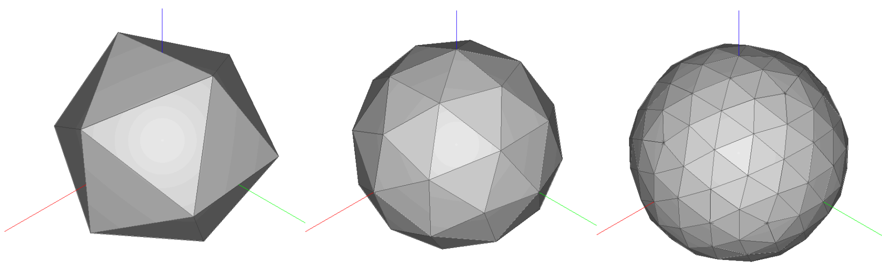

Fig. 5.1 (a) Icosahedron; (b) icosphere with 42 vertices; (c) icosphere with 162 vertices.

Multiple cell objects

The functions `INTERVALS` or `QUOTE` may be used to create many types and patterns of grid geometries.

Script 5.1.1 (Building frame)
First we give the main dataset of a building frame, by "quoting" the side measures of 2D design plan:

```
# Longitudinal trusses
Y = QUOTE([0.3, -6, 0.3, -6, 0.3])
# transverse beams
X = QUOTE([0.3, -3, 0.3, -4.2, 0.3, -3, 0.3])
# vertical measurements
Z = QUOTE([3,0.3])
```

Then, the alternate set of `INTERVALS` vector parameters are generated by Julia broadcast `.*` of the scalar `-1`, in order to invert all the signs.

```
X1 = QUOTE([0.3, -3, 0.3, -4.2, 0.3, -3, 0.3].* -1)
Y1 = QUOTE([0.3, -6, 0.3, -6, 0.3].*-1)
Z1 = QUOTE([3,-0.3].*-1)
```

Then the 3D building subsystems are generated:

```
# Cartesian product
pillars = COLOR(RED)(X*Y*Z);
trusses = COLOR(YELLOW)(X*Y1*Z1);
trusses1 = COLOR(YELLOW)(X1*Y*Z1);
floorslab = COLOR(GREEN)(X1*Y1*Z1);
```

Finally, the sub-complexes of 3D cells are aggregated in a single `Plasm` complex using the `STRUCT` combinator discussed in the next session:

```
frame = STRUCT(pillars, trusses, trusses1, floorslab);
VIEW(frame, Dict("background_color"=>WHITE))
```

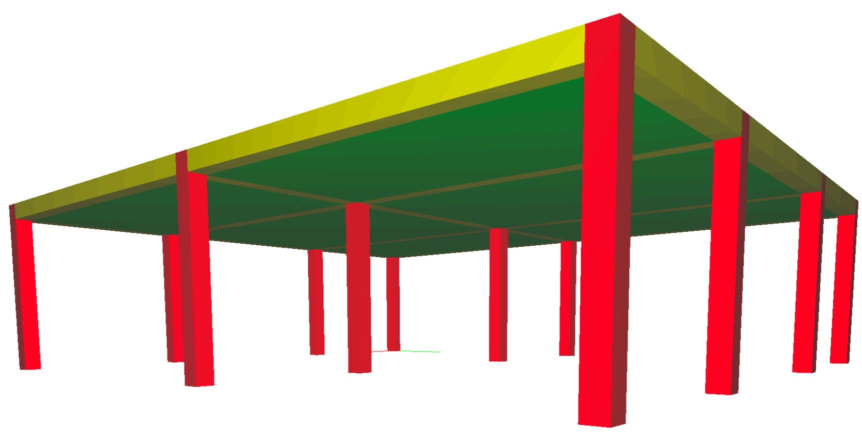

Fig. 5.2 frame = STRUCT(pillars, trusses, trusses1, floorslab);
.

Script 5.1.2 (Building skeleton)
We assemble here a building skeleton model, by creating a `STRUCT` assembly generated by `n=7`instances of the Julia `Vector` made by the `Hpc` value `frame` and by the `MatrixNd` value `T(3)(3.3)` producing a translation in z direction.

```
skeleton = STRUCT(NN(7)([frame, T(3)(3.3)]));
VIEW(skeleton, Dict("background_color"=>WHITE)
```

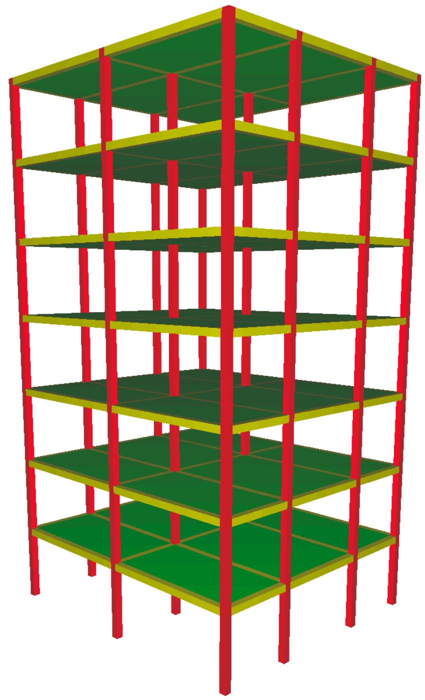

Fig. 5.3 skeleton = STRUCT(NN(7)([frame, T(3)(3.3)]));

.

Assembly aggregator STRUCT

5.2 Plasm topological operators

5.3 Linear and affine operators

5.4 Manifold mapping

5.5 Curve, surface, and solid methods

References

1. Arakaki., T.: Julia Data Parallel Computing. URL `https://juliafolds.github.io/data-parallelism/tutorials/quick-introduction/`. [retrieved june 27, 2023]
2. Arnold, D.N.: Finite Element Exterior Calculus, *CBMS-NSF Regional Conference Series in Applied Mathematics*, vol. 93. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (2018)
3. Aubanel, E.: Elements of Parallel Computing. Chapman Hall/CRC Press (2016)
4. Backus, J.: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Commun. ACM **21**(8), 613641 (1978). DOI 10.1145/359576.359579. URL `https://doi.org/10.1145/359576.359579`
5. Backus, J., Williams, J., Wimmers, E.: An introduction to the programming language FL. In: D. Turner (ed.) Research Topics in Functional Programming. Addison-Wesley, Reading, MA (1990)
6. Backus, J., Williams, J.H., Wimmers, E.L., Lucas, P., Aiken, A.: FL LANGUAGE MANUAL. PARTS l AND 2. Tech. Rep. RJ 7100 (67163), IBM Almaden Research Center (1989)
7. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1), 65–98 (2017). URL `https://doi.org/10.1137/141000671`
8. Carlsson, C.: Syntax highlighting – OhMyREPL. URL `https://kristofferc.github.io/OhMyREPL.jl/latest/`. [retrieved may 22, 2024]
9. Cimrman, R.: Sparse matrices in scipy. In: G. Varoquaux, E. Gouillart, O. Vahtras, P. deBuyl (eds.) Scipy lecture notes, release: 2022.1 edn., p. Section 2.5. Zenodo (2015). DOI 10.5281/zenodo.594102. URL `https://scipy-lectures.org/advanced/scipy_sparse/index.html`
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition, 3rd edn. The MIT Press (2009). URL `https://mitpress.mit.edu/9780262533058/introduction-to-algorithms/`
11. Danisc, S., Kavalar, M., Dombrowski, M., Markovics, P.: An Introduction to GPU Programming in Julia. URL `https://nextjournal.com/sdanisch/julia-gpu-programming`. [retrieved june 29, 2023]
12. Delfinado, C., Edelsbrunner, H.: An incremental algorithm for betti numbers of sinplicial complexes on the 3-sphere. Computer Aided Geometric Design **12**, 771–784 (1995)
13. DiCarlo, A., Milicchio, F., Paoluzzi, A., Shapiro, V.: Chain-based representations for solid and physical modeling. Automation Science and Engineering, IEEE Transactions on **6**(3), 454 –467 (2009). DOI 10.1109/giorgio
14. DiCarlo, A., Paoluzzi, A., Shapiro, V.: Linear algebraic representation for topological structures. Computer-Aided Design **46**, 269–274 (2014). DOI 10.1016/j.cad.2013.08.044. URL `https://doi.org/10.1016/j.cad.2013.08.044`
15. DiCarlo, A., Paoluzzi, A., Shapiro, V.: Linear algebraic representation for topological structures. Comput. Aided Des. **46**, 269–274 (2014). DOI 10.1016/j.cad.2013.08.044. URL `http://dx.doi.org/10.1016/j.cad.2013.08.044`
16. Ferrucci, V.: Generalised extrusion of polyhedra. In: Proceedings on the Second ACM Symposium on Solid Modeling and Applications, SMA '93, p. 3542. Association for Computing Machinery, New York, NY, USA (1993). DOI 10.1145/164360.164376. URL `https://doi.org/10.1145/164360.164376`
17. Ferrucci, V., Paoluzzi, A.: Extrusion and boundary evaluation for multidimensional polyhedra. Comput. Aided Des. **23**(1), 4050 (1991). DOI 10.1016/0010-4485(91)90080-G. URL `https://doi.org/10.1016/0010-4485(91)90080-G`
18. Hatcher, A.: Algebraic topology. Cambridge University Press (2002)

19. Julia: Manual: Base/lib-collections/iteration. URL https://docs.julialang.org/en/v1/base/collections/#lib-collections-iteration. [retrieved june 25, 2023]
20. Julia: Manual: Distributed-Computing. URL https://docs.julialang.org/en/v1/manual/distributed-computing/#Multi-processing-and-Distributed-Computing. [retrieved june 29, 2023]
21. Julia: Manual: Linear Algebra. URL https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#man-linalg. [retrieved july 3, 2023]
22. Julia: Manual: LinearAlgebra.factorize. URL https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#LinearAlgebra.factorize/. [retrieved june 25, 2023]
23. Julia: Manual: Metaprogramming. URL https://docs.julialang.org/en/v1/manual/metaprogramming/. [retrieved june 21, 2023]
24. Julia: Manual: Modules. URL https://docs.julialang.org/en/v1/manual/modules/#modules. [retrieved june 29, 2023]
25. Julia: Manual: Multi-Threading. URL https://docs.julialang.org/en/v1/manual/multi-threading/#man-multithreading. [retrieved june 29, 2023]
26. Julia: Manual: Parallel Computing. URL https://docs.julialang.org/en/v1/manual/parallel-computing/#Parallel-Computing. [retrieved june 26, 2023]
27. Julia: Manual: Parallel Computing. URL https://docs.julialang.org/en/v1/base/parallel/#Tasks. [retrieved june 27, 2023]
28. Julia: Manual: stdlib/SparseArrays. URL https://docs.julialang.org/en/v1/stdlib/SparseArrays/#man-csc/. [retrieved june 25, 2023]
29. Julia: Manual: Types. URL https://docs.julialang.org/en/v1/manual/types/#man-types. [retrieved june 30, 2023]
30. Julia Community: How to develop a Julia package. URL https://julialang.org/contribute/developing_package/. [retrieved june 29, 2023]
31. JuliaGPU: A gentle introduction to parallelization and GPU programming in Julia. URL https://cuda.juliagpu.org/stable/tutorials/introduction/#Introduction. [retrieved june 29, 2023]
32. Kamiski, B.: Julia for Data Analysis. Manning (2023)
33. Knuth, D.E.: Literate programming. Comput. J. **27**(2), 97–111 (1984). DOI 10.1093/comjnl/27.2.97. URL https://doi.org/10.1093/comjnl/27.2.97
34. Knuth, D.E.: Literate programming. In: SLI Lecture Notes, 27. Center for the Study of Language and Information (1992). URL https://web.stanford.edu/group/cslipublications/cslipublications/site/0937073806.shtml
35. Mainon, P.: Writing type-stable julia code (2021). URL https://blog.sintef.com/industry-en/writing-type-stable-julia-code/
36. Paoluzzi, A.: Geometric Programming for Computer Aided Design. John Wiley Sons, Chichester, UK (2003). URL https://onlinelibrary.wiley.com/doi/book/10.1002/0470013885
37. Paoluzzi, A., Pascucci, V., Vicentino, M.: Geometric programming: a programming approach to geometric design. ACM Trans. Graph. **14**(3), 266–306 (1995). DOI 10.1145/212332.212349. URL http://doi.acm.org/10.1145/212332.212349
38. Paoluzzi, A., Scorzelli, G., Vicentino, M.: Securing the cultural heritage via geometric programming and modeling. Tech. rep., Dept of Computer Science and Engineering, Roma Tre University (2009). URL https://www.academia.edu/47017676
39. Paoluzzi, A., Shapiro, V., DiCarlo, A., Furiani, F., Martella, G., Scorzelli, G.: Topological computing of arrangements with (co)chains. ACM Trans. Spatial Algorithms Syst. **7**(1) (2020). DOI 10.1145/3401988. URL https://doi.org/10.1145/3401988

40. Paoluzzi, A., Shapiro, V., DiCarlo, A., Scorzelli, G., Onofri, E.: Finite algebras for solid modeling using julias sparse arrays. Computer-Aided Design **155**, 103436 (2023). DOI https://doi.org/10.1016/j.cad.2022.103436. URL https://www.sciencedirect.com/science/article/pii/S0010448522001695
41. Permutohedron: Permutohedron — Wikipedia, the free encyclopedia (2023). URL https://en.wikipedia.org/wiki/Permutohedron. [Online; accessed 31-May-2024]
42. Roth, A., Weisstein, E.W.: Standard basis. In: MathWorld, p. Algebra > Linear Algebra > Linear Systems of Equations. A Wolfram Web Resource (2005). URL https://mathworld.wolfram.com/StandardBasis.html
43. Rowland, T.: Characteristic function. In: From MathWorld, p. Foundations of Mathematics > Set Theory > Sets. A Wolfram Web Resource (2005). URL ttps://mathworld.wolfram.com/CharacteristicFunction.html
44. Scorzelli, G.: Pyplasm library (2023). URL https://libraries.io/pypi/pyplasm
45. Scorzelli, G., Paoluzzi, A.: Plasm.jl: v0.1.0 (2023). URL https://github.com/scrgiorgio/Plasm.jl
46. Weisstein, E.W.: Julia set. From MathWorld–A Wolfram Web Resource URL https://mathworld.wolfram.com/JuliaSet.html
47. Whitaker, S.: Mastering the Julia REPL. URL https://blog.glcs.io/julia-repl#heading-starting-the-julia-repl. [retrieved may 22, 2024]
48. Whitney, H.: Geometric Integration Theory. Princeton University Press, Princeton (1957). DOI doi:10.1515/9781400877577. URL https://doi.org/10.1515/9781400877577
49. Wikibooks: Introducing julia/dictionaries and sets — wikibooks, the free textbook project (2020). URL https://en.wikibooks.org/. [Online; accessed 20-June-2023]
50. Williams, J.H., Wimmers, E.L.: Sacrificing Simplicity for Convenience: Where Do You Draw the Line? In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88, p. 169179. Association for Computing Machinery, New York, NY, USA (1988). DOI 10.1145/73560.73575. URL https://doi.org/10.1145/73560.73575